Popa & Weaver
Spring 2019

CS 161
Computer Security

Project 2

An End-to-End Encrypted File Sharing System

Author(s): You

This is the instruction version 6.
This version will be distributed by the end of Monday Feb 25, incorporating what happens in the OHs.

If today is after Mar 03, please visit the website for the latest version of the instructions.

Update: In version 6, we added a new requirement that filenames must be hidden (but the length can be exposed). See security guarantees (§4) and StoreFile (§4.3) for more information, and we clarified that we assume user passwords have sufficient entropy in InitUser (§4.1).

Update: In version 4, we changed the specification of RevokeFile (§5.3). We need to recreate the file with the same data, and we only assume the owner of the file will call this function.

Abstract: We want you to design and implement a file sharing system (*e.g.*, Dropbox) that protects user privacy. In particular, user files are always *encrypted* and *authenticated* on the server. In addition, users can *share* files with each other. These are basic functionalities for file sharing.

That being said, such an end-to-end encrypted file sharing system **won't be easy**. We urge you to start this project as early as possible.

Even with the help of a sample solution, two TAs spent **11 days** to finish this system, believe it or not!

Logistics:

- **Team size:** Up to two students.
- **Due date:** March 11 2019, 11:59 pm.

Roadmap:

- **Setup:**
  - □ Form a team.
  - □ Environment setup (§1).
    - □ Golang Tutorial.
    - □ Golang Quiz (§1.1).
    - □ Download the skeleton code (§1.2).

- □ Review the autograder rule (§1.3).

◇ **API Warmup:**

- □ UUID (§2.1)
- □ Datastore API (§2.2).
- □ Keystore API (§2.3).
- □ Cryptography functions (§2.4).

◇ **Preparation:**

- □ Recommended practice (§3).
- □ Skim the design document requirement (§6).

◇ **Part 1:**

- □ InitUser (§4.1) and GetUser (§4.2).
- □ LoadFile (§4.4) and StoreFile (§4.3).
- □ AppendFile (§4.5).
- □ Surprise break (§4.6).

◇ **Part 2:**

- □ ShareFile (§5.1) and ReceiveFile (§5.2).
- □ RevokeFile (§5.3).

◇ **Wrap up the submission:**

- □ Design document (§6).
- □ Submission (§7).

1 Environment Setup

The setup consists of five steps.

- □ Install Golang (link).
- □ Complete the online Golang Tutorial (link).
- □ Complete the Golang Quizzes (§1.1).
- □ Download the skeleton code (§1.2).
- □ Review the autograder rule (§1.3).

1.1 Golang quizzes

The following five questions will help you refresh the Golang Tutorial.

Quiz one: What does the ":=" mean in "x := 5"?

Quiz two: If an identifier in Golang starts with a non-capitalized letter, will this identifier be exported *i.e.*, accessible from outside this package?

Quiz three: How is a string converted into a byte slice?

Quiz four: How is the user structure converted into a byte slice? How is it recovered back from the byte slice?

Quiz five: Which website provides detailed documents for many Golang libraries?

1.2 Skeleton code

Skeleton code. You will be using the following template for this project:

https://inst.eecs.berkeley.edu/~cs161/sp19/proj2.tar

All of your code should go to `proj2.go` and `proj2_test.go`.

- `proj2.go`
  Where you will write your implementation.
- `proj2_test.go`
  Where you will write the tests. You must add tests to this file. The autograder will check the completeness of your test suite.

User library. You can access some useful functions in `userlib.go`, which you need to fetch with:

```
go get -u github.com/nweaver/cs161-p2/...
```

You can get familiar with these functions by reading §2, or by reading the source code:

https://github.com/nweaver/cs161-p2/

1.3 Autograder rule

Rule 1: No adversarial behavior. Your submission will be executed by the autograder on a series of tests. The autograder rejects any submission that imports new libraries or attempts anything similar to these:

- Spawn other processes.
- Read or write to the file system.
- Create any network connections.
- Attack the autograder by returning long, long output.

Code will be run in an isolated sandbox. Any adversarial behavior will be seen as cheating.

Rule 2: No global variables. Do not use global variables in `proj2.go`. All the functions you write in `proj2.go` must be stateless. That is to say, put data that needs permanent storage in the server stores, Keystore (§2.3) or Datastore (§2.2).

Rule 3: Return `nil` as the error code if an operation succeeds. Return a value different from `nil` as the error code if an operation fails. Many functions in this library have an output `error` for the error code. Please return `nil` if and only if there is no error. The autograder may decide whether the function is successful depending on whether the function returns an `nil` as the error code.

2 API

In this section we introduce some useful functions, including Google UUID (§2.1), Datastore (§2.2), Keystore (§2.3), and a few cryptographic functions (§2.4).

2.1 Universally unique identifier (UUID)

UUIDs are like unique names. As we will describe in §2.2, the Datastore requires us to use UUIDs as the key.

You will be using Google UUID library, the documentation of which can be found here. Below we outline how to randomly generate a UUID and how to deterministically generate a UUID from a byte slice.

Random sampling a UUID. Suppose you want to obtain a random UUID, we write:

```
new_UUID := uuid.New()
```

Deterministic encoding to a UUID. To convert a byte slice with ≥ 16 bytes into a UUID, we write:

```
(new_UUID, _) := uuid.FromBytes(byte_slice[:16])
```

which only takes the first 16 bytes.

Warning: The encoding does not hide the information in the byte slice. That is to say, you should not pass a confidential secret key as the byte slice here and use the resulting UUID publicly; an attacker may figure out information about the secret key from that UUID.

2.2 Datastore: A store of encrypted data

You place encrypted file data (and other metadata, as required by your design) on this server.

This server is **untrusted**, discussed in §4; thus, you must guarantee the **confidentiality** and **integrity** of any sensitive data or metadata you store on it.

You can use the following three API functions in `userlib`.

- `DatastoreSet(key UUID, value []byte)`.
  - Store `value` at `key`.
- `DatastoreGet(key UUID) (value []byte, ok bool)`.
  - Return the value stored at `key`.
  - If the key does not exist, `ok` will be false.
- `DatastoreDelete(key UUID)`.
  - Delete that item from the data store.

Note that the storage server has one namespace and does not do access control, so anything written by one user can be read or overwritten by any other user who knows the `key`. The client must ensure that their own files are not overwritten by other clients by using unique UUIDs that only this client knows.

2.3 Keystore: A store of public keys

You place your keys to a **trusted** public key server, that allows us to post and get public keys. You may need to post more than one key to the Keystore. There are two functions:

- `KeystoreSet(key string, value PKEEncKey/DSVerifyKey) error`.
  - Set the entry for `key` to be `value`.
  - Cannot modify an existing key-value entry, which will trigger an error.
- `KeystoreGet(key string) (value PKEEncKey/DSVerifyKey, ok bool)`.
  - Return the public key under the entry indexed by `key`.
  - If the key does not exist, `ok` will be false.

You can assume no attacker will overwrite any entry you add to the Keystore.

2.4 Cryptography functions

There are some cryptographic functions you can use. Note that if you use external cryptography libraries (or any other libraries), the autograder will refuse that and you will **get a zero**.

☐ I understand that I cannot use external cryptography libraries, or any other libraries.

☐ I understand that the autograder will refuse external libraries except those given in the skeleton (which includes `strconv`).

2.4.1 Public key encryption (PKE)

Data types:

- `PKEEncKey`: The encryption key (or the public key) for a public-key encryption.
- `PKEDecKey`: The decryption key (or the private key) for a public-key encryption.

Functions:

- `PKEKeyGen() (PKEEncKey, PKEDecKey, error)`.
  - Generate a RSA key pair for public-key encryption.
- `PKEEnc(ek PKEEncKey, plaintext []byte) ([]byte, error)`.
  - Use the RSA public key to encrypt a message.
  - *Warning:* It does not support very long plaintext. See here for a discussion.
- `PKEDec(dk PKEDecKey, ciphertext []byte) ([]byte, error)`.
  - Use the RSA private key to decrypt a message.

2.4.2 Digital signatures (DS)

Data type

- `DSSignKey`: The signing key (or the private key) for a digital signature scheme.
- `DSVerifyKey`: The verifying key (or the public key) for a digital signature scheme.

Functions:

- `DSKeyGen() (DSSignKey, DSVerifyKey, error)`.
  - Generate a RSA key pair for digital signatures.
- `DSSign(sk DSSignKey, msg []byte) ([]byte, error)`.
  - Use the RSA private key to create a signature.
- `DSVerify(vk DSVerifyKey, msg []byte, sig []byte) error`.
  - Use the RSA public key to verify a signature.

2.4.3 Hash-based message authentication code (HMAC)

HMAC generates a MAC given a 128-bit symmetric key and the byte slice. If used properly, it can be used to provide data integrity.

Functions:

- `HMACEval(key []byte, msg []byte) ([]byte, error)`.
  - Compute a SHA-512 HMAC of the message.
- `HMACEqual(a []byte, b []byte) bool`.
  - Compare whether two MACs are the same. Although `bytes.Equal` has the same functionality, the latter is not constant-time.

Warning: One key per purpose. If you use a key for symmetric encryption or HKDF, you should better not use this key for HMAC.

2.4.4 Hash-based key derivation function (HKDF)

HMAC can also be used as a simple hash-based key derivation function. HKDF allows you to derive a 128-bit symmetric key from a previous 128-bit symmetric key. If used properly, it can simplify key management.

Function:

- `HMACEval(key []byte, msg []byte) ([]byte, error)`.
  - You can use `HMACEval` and you **additionally** do a postprocessing step to take the first 16 bytes of the output as the symmetric key.

Warning: One key per purpose. If you use a key for symmetric encryption or HMAC, you should better not use this key for HKDF.

2.4.5 Password hashing function

Password hashing functions, also known as password key derivation functions, are commonly used to generate keys from passwords with some sort of entropy.

Function:

- `Argon2Key(password []byte, salt []byte, keyLen uint32) []byte`.
  - Output some bytes that can be used for symmetric keys. The size of the output equals `keyLen`.
  - *Warning:* You will need to use the salt in some way, so that even if the password is the same, the result should be different.

As the name implies, the password hashing function is the appropriate one to derive key from a password, where the password may only have some *middle* level of entropies. You should not (and will not be able to) use HKDF to derive a key from the password.

2.4.6 Symmetric encryption

Function:

- `SymEnc(key []byte, iv []byte, plaintext []byte) []byte`.
  - Encrypt using the CTR mode the plaintext using the key and an IV and output the ciphertext. The ciphertext will contain the IV, so you don't need to store the IV separately.
- `SymDec(key []byte, ciphertext []byte) []byte`.
  - Decrypt the ciphertext using the key.

Warning: One key per purpose. If you use a key for HKDF or HMAC, you should better not use this key for symmetric encryption.

Warning: The encryption uses the CTR mode. You should better supply a random IV.

Warning: Encryption does not promise integrity.

2.4.7 Random byte generator

Function:

- `RandomBytes(bytes int) (data []byte)`.
  - Given a length of random bytes you want, return the random bytes.
  - Can be used for IV or random keys.

Warning: If you use this function to generate random keys, you need to save them somewhere; otherwise, you will be unable to get them back.

2.4.8 Authenticated encryption

It will greatly simplify the design if you implement an authenticated encryption scheme here, which will be using HMAC, HKDF, and symmetric encryption in an interesting way.

But `userlib` intentionally did not implement authenticated encryption, not even using a mode of operation that simulatenously provides confidentiality and integrity (*e.g.*, GCM, OCB, EAX).

It is not required for us to write separate functions for authenticated encryption (and its decryption). But it is strongly recommended. When you feel that the code to provide simultaneously confidentially and integrity is too lengthy, you may come back here and implement authenticated encryption.

3 Design Security in From the Start

It is recommended to do three things iteratively.

Writing short sentences for the design document (DD).

- Create a Google document in Google Drive, shared with your teammate.
- Section 1 of the DD has been specified in §6. Copy those questions to the Google document.
- As you are working on the project, try to write down the design choices you make in the DD.
- Use the comment/assignment feature of Google document to nudge your teammate to do something.

Harley Patton's maxim: Come up with answers to questions for DD's Section 1 before you begin coding. Otherwise, you risk having to redo the entire project after discovering a security flaw in your design during the write-up.

Writing the `proj2.go` following the instructions.

- Printing out the instructions and using a pen to circle those that have been done may be useful.
- The TAs might or might not distribute limited printed copies of the instructions during the discussion sections and office hours.
- If you need to **compile** your code, see the next section on how to test.

Running and writing small tests for the project.

- Open `proj2_test.go` which already contains some basic tests.
- When you finish some code, run the following command in the terminal:

```
go test -v
```

- You are expected to see some syntax errors (if not, you are awesome). Fix them and continue.
- Add more tests for the things you are not sure about.
- It is normal that you failed some tests because you haven't yet implemented some functions. Don't be frustrated. We hope such error messages can encourage you to finish this project asap.

You should add more tests to `proj2_test.go` because the project will be graded based on the code coverage.

Please find how to test the code coverage in Golang here.

How to complete this project in one day?

It is recommended to find some ***consecutive***, rather than ***discrete***, hours ***with or without*** your teammate to work on the project 2. This is a big system and it will require a big STATE in your brain.

That said, please do not overstay during the office hours. Office hours also subject to the Fire Code.

You probably need a kettle of Earl Grey tea and a laptop charger.

4 Part 1: Single-User File Storage

System architecture. The file sharing system consists of many users and two servers.

- Many users connect to the two servers via a client program (designed by you!).
- The first server, Datastore, provides key-value storage for everyone.
- The second server, Keystore, provides a public key store for everyone.

You need to implement the stateless client program for the users.

Security guarantees:

- Any data placed on the servers should be available only to the owner and people that the owner explicitly shared the file with, not the server.
  - The server and other users should not learn the file content, the filename, and who owns the file, unless one shares the file with them.
- The server is allowed to know the length of the file content you store, which you don't need to hide.
- If the server modified the data you stored, you should be able to detect that and trigger an error. It is okay if the server returns a previous version of the file, which the client does not need to detect.
- ***New:*** Filenames must be hidden from the server, but it is okay to leak the length of the filenames.

Roadmap:

☐ Understand that you need to simultaneously provide confidentiality and integrity.

☐ Understand that you should not import any other libraries except those already in `proj2.go`.

☐ Understand that you should not create new global variables. All functions should run statelessly.

☐ User structures, **InitUser** (§4.1) and **GetUser** (§4.2). Here, the GetUser should be able to obtain the private key that you generated during the InitUser.

☐ Adding file storage functionalities, **LoadFile** (§4.4) and **StoreFile** (§4.3). Note that adding something in the `User` structure may simplify the design.

☐ Support efficient append, **AppendFile** (§4.5). This will likely modify your previous design of file storage.

4.1 InitUser: Create a user account.

Implement this function:

```
InitUser(username string, password string) (userdataptr *User, err error).
```

This function should:

- Generate the user data structure.
- Generate a key pair for digital signatures.
  - *i.e.*, the signing key (sk) and the verifying key (vk).

- Generate a key pair for asymmetric encryption.
  - *i.e.*, the decryption key (dk) and the encryption key (ek).
- Store private keys in the user structure.
- Store the user structure somewhere persistently. Note that you need confidentiality and integrity against the Datastore server (see §2.4 for some options).
- Post the two public keys somewhere publicly, so that other users can retrieve this user's public key.
- Return a pointer to this structure.
- See the footnote.[1]
- If the RSA key generation or any other functions you are using fails, return its error code. The autograder checks for the error code.
- If the function ends successfully, be sure to return `nil` for the error code.

New: We can assume that user passwords have sufficient entropy. Thus, it is infeasible for an attacker to guess a password. Yet, it is possible that two honest users choose the same password.

Warning: The client is stateless, meaning that the client forgets everything after system reboot. So, you need to ensure that given the same username and the same password, the user can later run GetUser to obtain the same private keys generated here.

4.2 GetUser: Log in using username and password.

Implement this function:

```
GetUser(username string, password string)
```

This function should:

- Obtain back the user data structure being created during InitUser, which is stored somewhere with some confidentiality and integrity.
- Obtain back the signing key for digital signatures.
- Obtain back the decryption key for asymmetric encryption.
- Return a pointer to this structure.
- If the data obtained is incorrect, for example, the integrity check fails, returns an error.
- If the function ends successfully, be sure to return `nil` for the error code.

Warning: You cannot use global variables to store user information, which will make the autograder unhappy. Like what was said in the InitUser, you need to store the user structure somewhere persistently (see §2 for some options).

[1] The secret shibboleth that you need to use during the office hours is: Diffie-Hellman. TAs may ask you this shibboleth to see if you actually read the instructions before coming to the office hours.

4.3 userdata.StoreFile: Store user files

Hint: Although intentionally not mentioned in InitUser and GetUser, it might be a good idea for us to first set up a file allocation table or something.

Implement this method:

```
StoreFile(filename string, data []byte)
```

This method should:

- Store the data for this file permanently on Datastore, with confidentiality and integrity guarantee.
- Must be place in a secret place in Datastore, otherwise other users may tamper with the data of the file, *e.g.*, remove the file content.
  - Note that the filename does not have sufficient entropy.
  - Different users should be allow to use the same filename and they should not interfere each other.
  - If two honest users have the same passwords, but they do not want to attack each other, their files should not interfere each other.
- ***New:*** Filenames must be hidden from the server, but it is okay to leak the length of the filenames.
- No need to return an error code. So if there is any error, ignore.
- *Warning:* If you change something in the user structure, don't forget to upload the new user structure – the user structure does not automatically synchronize with the remote version.
- *Recommendation:* You might want to generate the UUID used to store this filename from random and store this UUID somewhere, instead of deterministically derived from the filename. The same for the keys that you might use to encrypt and MAC the data. You will see why this is necessary when we go to Part 2 (§5). Please treat this recommendation seriously.

Warning: Each time you encrypt a file, you must use a different IV if you are using symmetric encryption.

4.4 userdata.LoadFile: Load user files

Implement this method:

```
LoadFile(filename string) (data []byte, err error)
```

This method should:

- Return the latest version of the file data.
- If the file does not exist, return `nil` as the data and trigger an error.
- If the file seems to be tampered, return `nil` as the data and trigger an error.
- Only return some non-`nil` data if it is the correct copy, which should pass some integrity check.

4.5 userdata.AppendFile: *Efficiently* append data to an existing file

Implement this method:

```
AppendFile(filename string, data []byte) (err error)
```

This method should:

- Append the data to the end of the file.
- If the file does not exist, return `nil` as the data and trigger an error.
- Do not need to check the integrity of the existing file. But if the file is badly broken, return `nil` as the data and trigger an error. This is not required.
- *Importantly*, this append must be **efficient**, defined as follows:
  - If before the append, the file has a size 1000TB, and you just want to add one byte, you should not need to download or decrypt the whole file. It should be *as lithe as a feather*.
  - Almost all of us reading this line will need to sit back and redesign how we store the file, which is expected. Don't panic. It is good.
  - You probably need to design some fancy file storage structure.
  - *Warning:* That said, you should treat cryptography as black boxes. Do not use cryptography *in a fancy way*, like playing with the IV value. **Instead**, this fancy file storage structure we mentioned here will be something else.
- Do not forget to update the user structure if you change it.

You need to write your own tests for AppendFile.

Warning: Recall that it is okay for the server to return *a previous version* of the file, but **not** a version that likely mixes the old and new data, as the security guarantees (§4) said. Double check whether your new design achieves this guarantee.

4.6 Before reaching the next stage

Congratulations! You deserve a pizza, good for you – Drew Barth.

Before you go to the next part, please do the following:

- Thank your teammate.
- Run `go test -v` in case there are some issues for the first two tests.
- Stay hydrated.
- Write some sentences in the DD (§3 and §6).
- Write some additional tests, if there are any need to increase the code coverage (see §3 for how to measure the code coverage).
- Submit a response to this Google form: link.

- Please answer one student question about the project 2 in Piazza.[2] But don't reveal too much. That being said, don't intentionally mislead someone else to the wrong direction (*e.g.*, TA told me that AES is broken last week, so don't use that).

[2] In case you don't know the link: https://piazza.com/class/jqemlqsll3a5hm

5 Part 2: Sharing and Revocation

System architecture, additional information: The file sharing system allows users to share files.

- Suppose there are two users U and V. They should have already posted their public keys somewhere so each one of them can find each other's public keys.
- If U wants to share file F with V, U assembles something we call *a magic string* and sends this magic string to V.
- Using this magic string, V can obtain full permission to F, allowing V to read, write, and share the file.
- U can revoke the permission for F from all other users.

Security guarantee, additional information:

- The channel that U and V use to talk to each other is insecure. Thus, that magic string needs both confidentiality and integrity. You can look for some options in §2.4.

Reminder of a recommendation:
We provided a recommendation in the description of StoreFile (§4.3). If you encounter major design difficulty in ShareFile (§5.1) or ReceiveFile (§5.1), come back to read that recommendation. Some substantial changes might be needed if you approach differs significantly from that recommendation.

API example: To help understand the functionality we want, here we show an example code:

```
u1, _ := GetUser("user_alice", "pw1")
u2, _ := InitUser("user_bob", "pw2")

v1, _ := u1.LoadFile("the_file_that_I_want_to_share_with_Bob")
magic_string, err := u1.ShareFile("the_file_that_I_want_to_share_with_Bob", "user_bob")

u2.ReceiveFile("the_file_from_alice", "user_alice", magic_string)
v2, _ = u2.LoadFile("the_file_from_alice")

// v1 should be the same as v2
```

As you can see here, after user Alice gave the magic string, user Bob can access the file (under a different filename, but actually the same file).

Importantly, if later Bob changes the file, Alice's file **WILL be UPDATED**. That is to say, Alice and Bob are actually sharing the same file, not just Alice sending a copy to Bob.

Roadmap:

- Sit back and think about how to implement that magic string.
- Implement ShareFile (§5.1) and ReceiveFile (§5.2) and run the test in `proj2_test.go`, as well as your additional tests.
- Implement RevokeFile (§5.3) and write more tests for the revoke functionality.

5.1 userdata.ShareFile: File sharing with confidentiality, integrity and authenticity

Implement this method:

```
ShareFile(filename string, recipient string) (magic_string string, err error)
```

This method should:

- Generate a magic string with some confidentiality, integrity and authenticity.
  - That is to say, only the recipient can use this magic string to obtain permission (confidentiality), and the recipient can verify whether the magic string is from the correct sender (authenticity) and has not been tampered with (integrity).
- If the file does not exist, trigger an error and return with an empty string.
- If you find it hard to convert something into a string, see here: link.
- The recipient should later be able to do all the following:
  - Read data in this file.
  - Write data to this file, which the sender will see the update.
  - Share this file with others.
  - Revoke this file, discussed later in §5.3.
- The recipient should only have permission about the file being shared, not other files from the sender. For example, sending the sender's password is not a valid solution.
- Do not assume the sender and the recipient are online at the same time. The permission must be passed in one shot, via the magic string.

5.2 userData.ReceiveFile: Adding file permission

Implement this method:

```
ReceiveFile(filename string, sender string, magic_string string) error
```

This method should:

- Create a file with the filename. Note that the filename does not need to be the same with the filename that the sender uses to call that file.
- If the filename has already been used in the recipient side, trigger an error and return.
- Do not forget to upload the user structure, if you change it.
- The recipient should be able to read, modify, share, and revoke this file as if he/she owns this file.
- The sender should see all the changes of the file. That is to say, there is only one file, and the file is really being shared.
- *Requirement:* Do not store the file multiple times. For example, if user Alice shares a 1000TB file with Bob, Bob should not create another file that takes another 1000TB storage space.

5.3 userData.RevokeFile: Burn it with fire

Read the title carefully, and implement this method:

```
RevokeFile(filename string) error
```

If user U owns file F calls this method on F, this method should:

- Delete file F, so that no other user can read this file.
  - If all other users sharing F did not save a local copy of the file content, nobody can obtain the content of this revoked file anymore.
  - Other users may see an error that the file does not exist or an empty file, you decide.
- Recreate file F with the same file content, but this time, not shared with anyone else.
- Return an error if F does not exist.

RevokeFile should work correctly when the owner U calls this method. If U shares this file with V, and user V invokes RevokeFile, the result is undefined. In particular, it is ok if your implementation allows V to successfully revoke the file so that neither U nor any other user can access it.

Example Here is an example workflow of RevokeFile, as follows.

If user Alice owns "file" and shared the file to Bob.

```
u1, _ := GetUser(" user_alice ", "pw1")
u2, _ := InitUser("user_bob", "pw2")

magic_string, err := u1.ShareFile(" file ", "user_bob")
u2.ReceiveFile(" alice_file ", " user_alice ", magic_string)
```

Then Bob will have access to Alice's file, under the filename "alice_file".

Next, Alice revokes the permission of this file, that is, deletes this file, and recreates it.

```
u1, _ := GetUser(" user_alice ", "pw1")
u1.RevokeFile(" file ")
```

Now, Alice can still access "file", but Bob cannot.

```
file_data , _ := u1.LoadFile(" file ")
```

Alice can update the file:

```
u1, _ := GetUser(" user_alice ", "pw1")
u1. StoreFile (" file ", new_file_data)
```

Alice no longer shares this file with Bob. You must ensure two things:

- Bob does not have access to the new file content.

- Bob does not even realize that the file is recently updated by Alice.

This property might require us change the implementation of key management, including StoreFile (§4.3), LoadFile (§4.4), and AppendFile (§5.1), which is expected if you did not follow the recommendation in §4.3. Basically, file location and file keys should better not depend on the filename.

6 Design document

Write a clear, concise design document (DD) to go along with your code. Your DD should be split into two sections. The first contains the design of your system, and the choices you made; the second contains a security analysis.

A well-written DD receiving full points could be even less than two pages! If a DD is excessively verbose[3], this DD might lose points.

You do not need to draw fancy figures in the DD because it is not worthwhile. A full-point DD only needs to explain the ideas clearly.

If you followed the recommendation in §3, the Google document should be a good start for your DD. You can create the PDF for DD in Google document by File ⟶ Download as ⟶ PDF Document (.pdf).

Section 1: System Design

In the first section, summarize the design of your system. Explain the major design choices you made, written in a manner such that an average 161 student could take it, re-implement your client, and achieve a grade similar to yours. It should also describe your testing methodology in your `proj2_test.go` file.

If you're looking for somewhere to get started, you can begin by asking yourselves six questions:

- □ How is each client initialized?
- □ How is a file stored on the server?
- □ How are file names on the server determined?
- □ What is the process of sharing a file with a user?
- □ What is the process of revoking a user's access to a file?
- □ How were each of these components tested?

Section 2: Security Analysis

The second part of your design document is a security analysis.

⋄ Present **at least three** and **at most five** concrete attacks that you have come up with and how your design protects against each attack.

Only three are needed for full credit; in the event that more than three are provided your score will be determined by the three which provide us the most credit.

You should not need more than one paragraph each to explain how your implementation defends against the attacks you present. Make sure that your attacks cover different aspects of the design of your system. That is, don't provide three attacks all involving file storage but nothing involving sharing or revocation.

[3] If after writing your design document, you realize that you have a 10-page document with 100 lines of code and think to yourselves "The CS162 TAs would be proud of this," you will likely be disappointed in your grade. That is not a design document. That is an implementation with comments.

7 Submission and Grading

Please submit a response to another Google Form (link) so the TAs can estimate how many people complete the project.

Your final score on this part of the project will be the minimum of the functionality score and security score. Each failed security test will lower the security score, weighted by the impact of the vulnerability.

All tests will be run in a sandbox, and if your code is in any way malicious, we will notice, as described in §1.3. Please be gentle to the autograder, which is going to having his first birthday.

8 Submission Summary

You must submit the following files for the project. See announcement in Gradescope for more information.

```
proj2.go
proj2_test.go
design.pdf
```